

## DRAGON BALL. Episodio 3

Después de que Jacu investigara el servidor de la empresa Capsule Corp con la ayuda del Dr. Brief, las sospechas de Trunks quedaron confirmadas. La existencia de un fichero sospechoso que no ha sido creado por nadie de la empresa, parece indicar que ésta ha sido comprometida. Para aclarar el enigma, el Dr. Brief pide a Bulma que estudie el fichero en cuestión y extraiga cualquier información relevante que pudiera arrojar luz sobre el caso. Bulma consigue obtener la información de su creador, el Dr. Raichi, y descubre, además, que el contenido del fichero está escrito en una extraña lengua de la raza Tsufur. Para descifrar el contenido y obtener el texto en un lenguaje que ellos comprendan necesitan una clave. ¿Podrás encontrarla?

Descarga del fichero:

[https://drive.google.com/file/d/1UihvI5nEjkarfM03DV8J5RgJ2ZD1zy\\_T/view?usp=sharing](https://drive.google.com/file/d/1UihvI5nEjkarfM03DV8J5RgJ2ZD1zy_T/view?usp=sharing)

Info: La flag tiene el formato UAM{md5 del string encontrado}

### Resolución

Descargamos el fichero **main**.

Analizamos

#### **file main**

```
main : ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=6c62b346db422600ce26c67127d49dbb77ba4878, not stripped
```

Probamos funcionamiento:

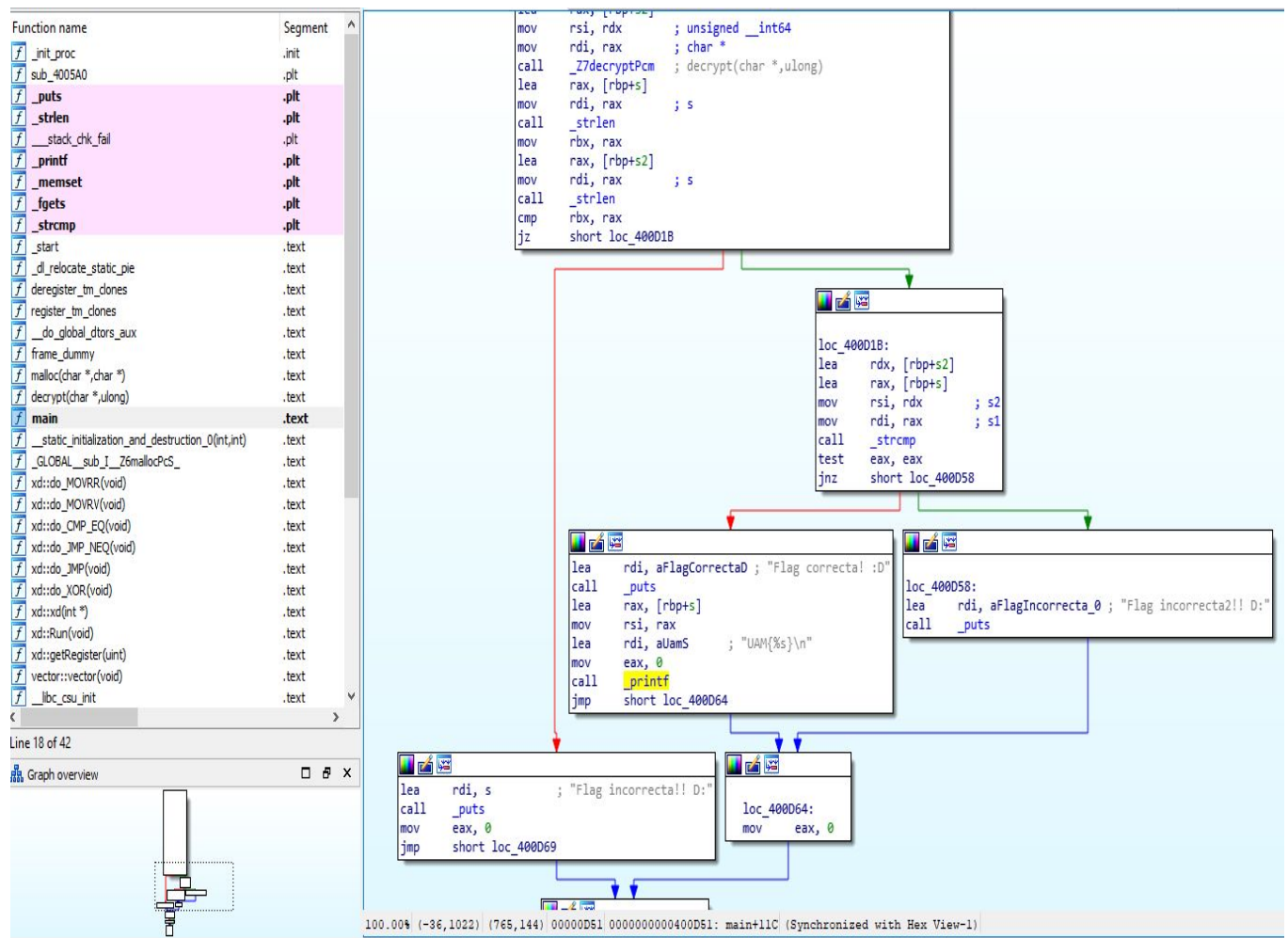
**./main**

Flag: 00000000

Flag incorrecta!! D:

Pasamos al análisis estático con la versión gratuita de Ida.

Empezamos por la función main, donde aparece la cadena ("flag incorrecta!! ")



La función main, nos muestra el flujo del programa, en el que observamos que para llegar a la flag correcta se hacen dos comprobaciones, una primera con el tamaño de la cadena `_strlen` y una segunda comparando dos cadenas `_strcmp`.

Estudiemos las cadenas comparadas `s`, `s2` (`rbp+s2` y `rbp+s`).

```

xor     eax, eax
lea     rdi, format      ; "Flag: "
mov     eax, 0
call    _printf
mov     rdx, cs:__bss_start ; stream
lea     rax, [rbp+s]
mov     esi, 14h         ; n
mov     rdi, rax         ; s
call    _fgets
lea     rax, [rbp+s]
mov     rdi, rax         ; s
call    _strlen
mov     [rbp+var_48], rax
mov     rax, [rbp+var_48]
sub     rax, 1
mov     [rbp+rax+s], 0
mov     [rbp+s2], 7
mov     [rbp+var_38], 59h
mov     [rbp+var_3A], 10h
mov     [rbp+var_39], 36h
mov     [rbp+var_38], 1Ah
mov     [rbp+var_37], 59h
mov     [rbp+var_36], 36h
mov     [rbp+var_35], 5Ah
mov     [rbp+var_34], 5Dh
mov     [rbp+var_33], 1Ah
mov     [rbp+var_32], 30h
mov     [rbp+var_31], 0
lea     rax, [rbp+s2]
mov     rdi, rax         ; s
call    _strlen
mov     rdx, rax
lea     rax, [rbp+s2]
mov     rsi, rdx         ; unsigned __int64
mov     rdi, rax         ; char *
call    _Z7decryptPcm    ; decrypt(char *,ulong)
lea     rax, [rbp+s]
mov     rdi, rax         ; s
call    _strlen
mov     rbx, rax
lea     rax, [rbp+s2]
mov     rdi, rax         ; s
call    _strlen
cmp     rbx, rax
jz      short loc_400D1B

```

La primera cadena, [rbp+s] es la que introducimos nosotros \_fgets.

La otra, se establece inicialmente en el código [7,59,1d,36,1a,59,36,5a,5d,1a,30], después se modifica con la función (**Z7decryptPcm**), donde pasamos como parámetros, la cadena y su tamaño \_strlen

```

; __int64 __fastcall decrypt(char *, unsigned __int64)
public _Z7decryptPcm proc near
_Z7decryptPcm
var_20= qword ptr -20h
var_18= qword ptr -18h
var_4= dword ptr -4
push    rbp
mov     rbp, rsp
mov     [rbp+var_18], rdi
mov     [rbp+var_20], rsi
mov     [rbp+var_4], 0

```

La parte interesante de esta función, es la utilización de xor 69h para toda la cadena (xor ecx, 69h).

[07,59,1d,36,1a,59,36,5a,5d,1a,30] ⊕ 69

[6e,30,74,5f,73,30,5f,33,34,73,59]

n0t\_s0\_34sY (11 caracteres 0b)

```

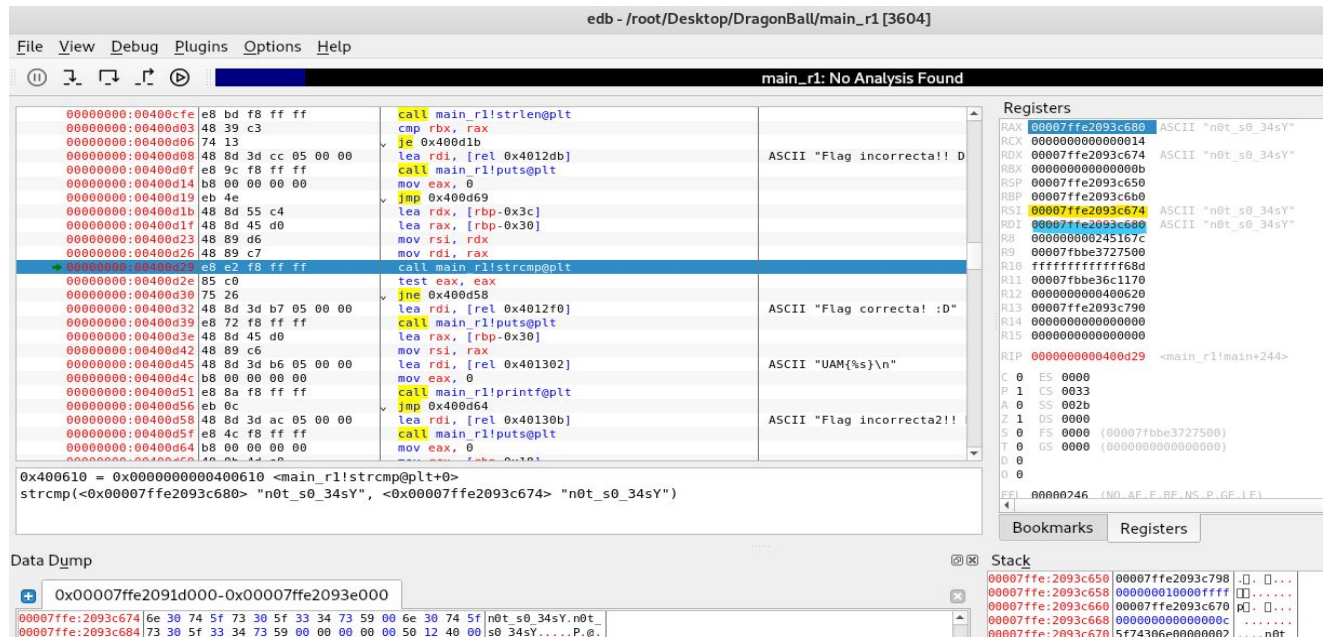
loc_400BFD:
mov     eax, [rbp+var_4]
cdqe
cmp     [rbp+var_20], rax
jbe     short loc_400C32

mov     eax, [rbp+var_4]
movsxd  rdx, eax
mov     rax, [rbp+var_18]
add     rax, rdx
movzx   ecx, byte ptr [rax]
mov     eax, [rbp+var_4]
movsxd  rdx, eax
mov     rax, [rbp+var_18]
add     rax, rdx
xor     ecx, 69h
mov     edx, ecx
mov     [rax], dl
add     [rbp+var_4], 1
jmp     short loc_400BFD

loc_400C32:
nop
pop     rbp
retn
_Z7decryptPcm endp

```

Tras unos cuantos “F8”, llegamos a la función main 00400c35, seguimos, introducimos cadena “n0t\_s0\_34sy”, seguimos hasta llegar a la comparación de cadenas.



En la imagen tenemos resaltado en amarillo la cadena s2 y en azul la s1. en el momento de la comparacion (strcmp). El programa finaliza correctamente mostrando "Flag correcta! :D"

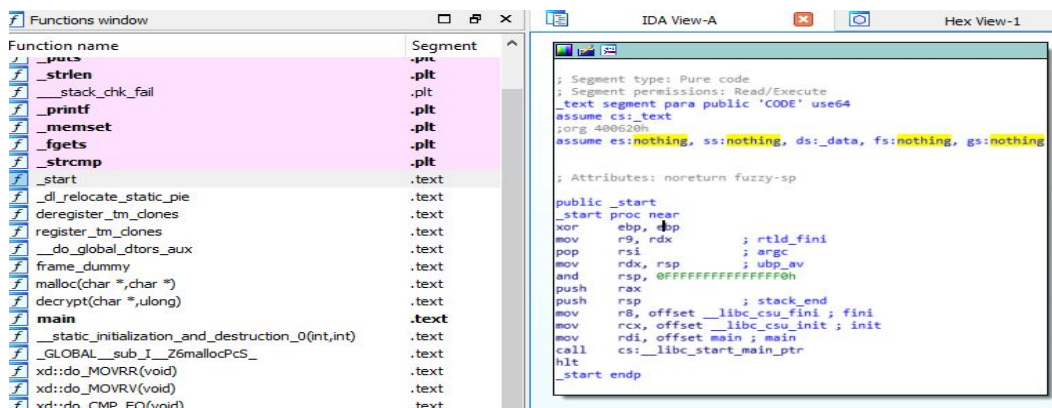
```
UAM{no t s0 34sy}
```

Demasiado fácil..... (sobra decir que no es la flag)

Tendremos que seguir buscando. Volvemos a Ida

Si nos fijamos, disponemos muchas funciones que a priori no se utilizan (`do_MOVR`, `do_JMP`.....). Tendremos que analizar desde el principio, a ver que se nos escapa...

Empezamos en el Entry Point del programa (función **\_start**) 0x400620h



Utiliza una llamada a `lib_c_start_main`, pasándole 3 parámetros (`main`, `csu_init`, `csu_finit`). La siguiente página, nos explica detalladamente su funcionamiento.  
<http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html>

Tenemos que `csu_init` es un puntero a la función que realiza la inicialización inicial (constructor) y `csu_fini` otro al que se encarga de la inicialización final (destructor).

Lo importante es que la llamada al constructor se realiza antes de “llamar” `main`.

Esta vez, en EDB, nos quedamos en 400620,

```

edb - /root/L
File View Debug Plugins Options Help
[Icons]
r12 00000000:00400620 31 ed xor ebp, ebp
00000000:00400622 49 89 d1 mov r9, rdx
00000000:00400625 5e pop rsi
00000000:00400626 48 89 e2 mov rdx, rsp
00000000:00400629 48 83 e4 f0 and rsp, 0xfffffffffffffff0
00000000:0040062d 50 push rax
00000000:0040062e 54 push rsp
00000000:0040062f 49 c7 c0 c0 12 40 00 mov r8, 0x4012c0
00000000:00400636 48 c7 c1 50 12 40 00 mov rcx, 0x401250
00000000:0040063d 48 c7 c7 35 0c 40 00 mov rdi, 0x400c35
00000000:00400644 ff 15 a6 19 20 00 call qword [rel 0x601ff0]
00000000:00400645 91 ret

```

En `rcx` tenemos el constructor (0x401250)

```

.text:0000000000401250 ; void __libc_csu_init(void)
.text:0000000000401250 public __libc_csu_init
.text:0000000000401250 __libc_csu_init proc near ; DATA XREF: __start+1610
.text:0000000000401250 push r15
.text:0000000000401252 push r14
.text:0000000000401254 mov r15, rdx
.text:0000000000401257 push r13
.text:0000000000401259 push r12
.text:000000000040125B lea r12, frame_dummy_init_array_entry
.text:0000000000401262 push rbp
.text:0000000000401264 lea rbp, __do_global_ctors_aux_fini_array_entry
.text:000000000040126A push rbx
.text:000000000040126B mov r13d, edi
.text:000000000040126E mov r14, rsi
.text:0000000000401271 sub rbp, r12
.text:0000000000401274 sub rsp, 8
.text:0000000000401278 sar rbp, 3
.text:000000000040127C call __init_proc
.text:0000000000401281 test rbp, rbp
.text:0000000000401284 jz short loc_4012A6
.text:0000000000401286 xor ebx, ebx
.text:0000000000401288 nop
.text:0000000000401290 loc_401290: ; CODE XREF: __libc_csu_init+544j
.text:0000000000401290 mov rdx, r15

.init_array:0000000000601E08 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:0000000000601E08 ; DATA XREF: LOAD:00000000004000F810
.init_array:0000000000601E08 ; LOAD:000000000040021010 ...
.init_array:0000000000601E08 ; Alternative name is '__init_array_start'
.init_array:0000000000601E10 dq offset GLOBAL__sub_I__Z6mallocPcS_
.init_array:0000000000601E10 __init_array ends

```

Ya tenemos nuestra función inicial. `__GLOBAL__sub_I__Z6mallocPcS_`

La secuencia de llamadas sería:

`__GLOBAL__sub_I__Z6mallocPcS_ -> _Z41__static_initialization_and_destruction_0ii -> _ZN6vectorC2Ev.`



```

; Attributes: bp-based frame

; __int64 __fastcall vector::vector(vector * __hidden this)
public _ZN6vectorC2Ev ; weak
_ZN6vectorC2Ev proc near

var_28= qword ptr -28h
var_1C= dword ptr -1Ch
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

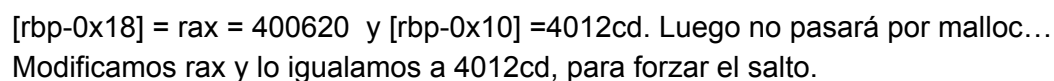
push     rbp                                ; Alternative name is 'vector::vector(void)'
mov      rbp, rsp
mov      [rbp+var_28], rdi
mov      [rbp+var_1C], 0
lea      rax, __start
mov      [rbp+var_18], rax
lea      rax, __etext
mov      [rbp+var_10], rax

loc_4011FB:
mov      rax, [rbp+var_18]
cmp      rax, [rbp+var_10]
jz       short loc_40122B

loc_40122B:
mov      [rbp+var_8], offset off_602048
lea      rdx, _Z6mallocPcS_ ; malloc(char *,char *)
mov      rax, [rbp+var_8]

```

Recargamos el ejecutable en EDB y ponemos punto de interrupción en `_ZN6vectorC2Ev`  
**0x4011D6**



Continuamos:

Address	Disassembly	Comment
00000000:0040121a	83 45 e4 01	add dword [rbp-0x1c], 1
00000000:0040121e	83 7d e4 06	cmp dword [rbp-0x1c], 6
00000000:00401222	77 1f	ja 0x401243
00000000:00401224	48 83 45 e8 01	add qword [rbp-0x18], 1
00000000:00401229	eb d0	jmp 0x4011fb
00000000:0040122b	48 c7 45 f8 48 20 60 00	mov qword [rbp-8], 0x602048
00000000:00401233	48 8d 15 cd f4 ff ff	lea rdx, [rel 0x400707]
00000000:0040123a	48 8b 45 f8	mov rax, [rbp-8]
00000000:0040123e	48 89 10	mov [rax], rdx
00000000:00401241	eb 01	jmp 0x401244
00000000:00401243	90	nop
00000000:00401244	5d	pop rbp
00000000:00401245	c3	ret
00000000:00401246	66 2e 0f 1f 84 00 00 0...	nop word cs:[rax+rax]
00000000:00401250	41 57	push r15
00000000:00401252	41 56	push r14

qword ptr [rax] = [0x0000000000602048] = 0x000000000400616  
rdx = 0x000000000400707

Register	Value
RAX	0000000000602048
RCX	00007f498fd72718
RDX	000000000400707
RBX	0000000000000001
RSP	00007ffea3a1d330
RBP	00007ffea3a1d330
RSI	000000000000ffff
RDI	0000000000602069
R8	00007f498fd73d80
R9	00007f498fd73d80
R10	0000000000000000
R11	0000000000000000
R12	0000000000601e08
R13	0000000000000001
R14	00007ffea3a1d488
R15	00007ffea3a1d498
RIP	000000000040123e

Aquí tenemos una parte de lo más interesante, lo que realiza es cargar en rdx el puntero a la función malloc (400707) y poner ese valor en 0x602048, y ¿esta dirección ?, la miramos en ida

**.got.plt:0000000000602048 off\_602048 dq offset strcmp ; DATA XREF: \_strcmp↑r**

Modifica el puntero a la función strcmp y lo sustituye por malloc, lo que quiere decir, que en nuestra función main, cuando se llame a strcmp, llamará a malloc.

Para verificarlo, seguimos depuración y ponemos punto de interrupción en 400d1b, ponemos como cadena "000000000000".

Address	Disassembly	Comment
00000000:00400d1b	48 8d 55 c4	lea rdx, [rbp-0x3c]
00000000:00400d1f	48 8d 45 d0	lea rax, [rbp-0x30]
00000000:00400d23	48 89 d6	mov rsi, rdx
00000000:00400d26	48 89 c7	mov rdi, rax
00000000:00400d29	e8 e2 f8 ff ff	call main!strcmp@plt
00000000:00400d2e	85 c0	test eax, eax

Register	Value
RAX	00007ffea3a1d370 ASCII "000000000000"
RCX	0000000000000004
RDX	00007ffea3a1d364 ASCII "\n0t_s0_34sY"
RBX	000000000000000b
RSP	00007ffea3a1d340

F7 -> F8 y BINGO!!, estamos en malloc 0x400707.

Llegado a este punto, para facilitar la depuración, y no tener que modificar rax, en \_ZN6vectorC2Ev, a mano para que realice el salto de cambiar el puntero a malloc, parcheamos el ejecutable.

Address	Disassembly	Comment
00000000:004011de	c7 45 e4 00 00 00 00	mov dword [rbp-0x1c], 0
00000000:004011e5	48 8d 05 34 f4 ff ff	lea rax, [rel 0x400620]
00000000:004011ec	48 89 45 e8	mov [rbp-0x18], rax
00000000:004011f0	48 8d 05 d6 00 00 00	lea rax, [rel 0x4012cd]
00000000:004011f7	48 89 45 f0	mov [rbp-0x10], rax
00000000:004011fb	48 8b 45 e8	mov rax, [rbp-0x18]
00000000:004011ff	48 3b 45 f0	cmp rax, [rbp-0x10]
00000000:00401203	74 26	je 0x40122b

La idea es cambiar la instrucción mov rax,[rbp-0x18] por mov rax,[rbp-0x10] con lo que siempre serán iguales en el cmp. Para esto con un editor hexadecimal buscamos la cadena

“48 89 45 f0 48 8b 45 **e8** 48 3b 45 f0” y la sustituimos por “48 89 45 f0 48 8b 45 f0 48 3b 45 f0”. Realmente es cambiar el valor **e8** [rbp-0x18] por **f0** [rbp-0x10].

00000000:004011e5	48 8d 05 34 f4 ff ff	lea rax, [rel 0x400620]
00000000:004011e6	48 89 45 e8	mov [rbp-0x18], rax
00000000:004011f0	48 8d 05 d6 00 00 00	lea rax, [rel 0x4012cd]
00000000:004011f7	48 89 45 f0	mov [rbp-0x10], rax
00000000:004011fb	48 8b 45 f0	mov rax, [rbp-0x10]
00000000:004011ff	48 3b 45 f0	cmp rax, [rbp-0x10]
00000000:00401203	74 26	je 0x40122b

Volvemos a la función malloc.

Tras analizarla, comprobamos que realiza una primera inicialización de valores, y llamadas a ciertas funciones `xd:xd`, `xd:run..` `xd::Movrr`, etc.

Parece que nos encontramos en una especie de VM, que ejecuta una serie de operaciones.

En `_ZN2xd3RunEv` ; `__int64 __fastcall xd::Run(xd *__hidden this)` podemos ver las diferentes llamadas condicionales a funciones, en base a un valor.(opcode)

Resumiendo tenemos:

**99h => RET (END)**

**55h => do\_MOVRR**

**33h => do\_CMP\_EQ**

**44h => do\_JMP\_NEQ**

**77h => do\_JMP**

**88h => do\_XOR**

**69h => do\_MOVRV**

Ahora debemos analizar la inicialización de la VM, malloc, que es donde se establece el funcionamiento.

EDB, punto de interrupción 0x400707 y luego ejecutamos hasta 400a99.

00000000:00400a84	c7 45 e4 13 00 00 00	mov dword [rbp-0x1c], 0x13
00000000:00400a8b	c7 45 e8 01 00 00 00	mov dword [rbp-0x18], 1
00000000:00400a92	c7 45 ec 99 00 00 00	mov dword [rbp-0x14], 0x99
00000000:00400a99	8b 85 68 fd ff ff	mov eax, [rbp-0x298]
00000000:00400a9f	89 85 e4 fd ff ff	mov [rbp-0x21c], eax
00000000:00400aa5	48 8b 85 58 fd ff ff	mov rax, [rbp-0x2a8]
00000000:00400aac	0f b6 00	movzx eax, [rax]
00000000:00400aad	0f be c0	movsx eax, al
00000000:00400ab2	89 85 04 fe ff ff	mov [rbp-0x1fc], eax
00000000:00400ab8	48 8b 85 58 fd ff ff	mov rax, [rbp-0x2a8]
00000000:00400abf	48 83 c0 01	add rax, 1
00000000:00400ac3	0f b6 00	movzx eax, [rax]
00000000:00400ac6	0f be c0	movsx eax, al
00000000:00400ac9	89 85 30 fe ff ff	mov [rbp-0x1d0], eax
00000000:00400acf	48 8b 85 58 fd ff ff	mov rax, [rbp-0x2a8]
00000000:00400ad6	48 83 c0 02	add rax, 2

rax = 0x00007ffe83285aa0

**Registers**

RAX	00007ffe83285aa0	ASCII "000000000000"
RCX	0000000000000000	
RDX	00007ffe83285830	
RBX	000000000000000b	
RSP	00007ffe832857b0	
RBP	00007ffe83285a60	
RSI	00007ffe83285a94	ASCII "n0t_s0_34sY"
RDI	00007ffe83285a50	
R8	000000000212067c	
R9	00007f3e82056500	
R10	ffffffffffff68d	
R11	00007f3e81ff0170	
R12	000000000400620	
R13	00007ffe83285bb0	
R14	0000000000000000	
R15	0000000000000000	
RIP	000000000400abf	<main_ri!malloc(char*, char*)+952>

Si seguimos la ejecución, vemos cómo se inicializa un vector con las instrucciones(opcodes) y después se utiliza la cadena “000000000000” y su tamaño, para insertarlas en diferentes zonas del vector, separando la cadena en caracteres individuales.



Aquí tenemos el volcado en memoria de las instrucciones de la VM, en amarillo está resaltado la parte de la cadena introducida por teclado ("000000000000" tamaño "0b").

00000000:00400ba3	48 05 00 00	mov r11, rax
00000000:00400ba4	e8 ad 04 00 00	call main_r1!xd::xd(int*)
00000000:00400bb1	48 8d 85 70 fd ff ff	lea rax, [rbp-0x290]
00000000:00400bb2	48 05 00 00	mov r11, rax

0x40105e = 0x000000000040105e <main\_r1!xd::xd(int\*)+0>

Data Dump

+ 0x00007ffcc2ca4000-0x00007ffcc2cc5000

00007ffc:c2cc28b0	77 00 00 00 03 00 00 00 99 00 00 00 69 00 00 00	w.....i...
00007ffc:c2cc28c0	00 00 00 00 0b 00 00 00 33 00 00 00 00 00 00 00	.....3.....
00007ffc:c2cc28d0	0b 00 00 00 44 00 00 00 02 00 00 00 69 00 00 00	.....i...
00007ffc:c2cc28e0	00 00 00 00 30 00 00 00 88 00 00 00 00 00 00 00	.....0.....
00007ffc:c2cc28f0	d2 00 00 00 33 00 00 00 00 00 00 00 95 00 00 00	.....3.....
00007ffc:c2cc2900	44 00 00 00 02 00 00 00 69 00 00 00 00 00 00 00	D.....i.....
00007ffc:c2cc2910	30 00 00 00 88 00 00 00 00 00 00 00 d6 00 00 00	.....0.....
00007ffc:c2cc2920	33 00 00 00 00 00 00 00 e6 00 00 00 44 00 00 00	3.....D.....
00007ffc:c2cc2930	02 00 00 00 69 00 00 00 00 00 00 00 30 00 00 00	.....i.....0...
00007ffc:c2cc2940	88 00 00 00 00 00 00 00 87 00 00 00 33 00 00 00	.....3.....
00007ffc:c2cc2950	00 00 00 00 d3 00 00 00 44 00 00 00 02 00 00 00	.....D.....
00007ffc:c2cc2960	69 00 00 00 00 00 00 00 30 00 00 00 88 00 00 00	i.....0.....
00007ffc:c2cc2970	00 00 00 00 ea 00 00 00 33 00 00 00 00 00 00 00	.....3.....
00007ffc:c2cc2980	b5 00 00 00 44 00 00 00 02 00 00 00 69 00 00 00	.....D.....i...
00007ffc:c2cc2990	00 00 00 00 30 00 00 00 88 00 00 00 00 00 00 00	.....0.....
00007ffc:c2cc29a0	d4 00 00 00 33 00 00 00 00 00 00 00 bc 00 00 00	.....3.....
00007ffc:c2cc29b0	44 00 00 00 02 00 00 00 69 00 00 00 00 00 00 00	D.....i.....
00007ffc:c2cc29c0	30 00 00 00 88 00 00 00 00 00 00 00 02 00 00 00	.....0.....
00007ffc:c2cc29d0	33 00 00 00 00 00 00 00 32 00 00 00 44 00 00 00	3.....2...D...
00007ffc:c2cc29e0	02 00 00 00 69 00 00 00 00 00 00 00 30 00 00 00	.....i.....0...
00007ffc:c2cc29f0	88 00 00 00 00 00 00 00 1b 00 00 00 33 00 00 00	.....3.....
00007ffc:c2cc2a00	00 00 00 00 2b 00 00 00 44 00 00 00 02 00 00 00	.....+...D.....
00007ffc:c2cc2a10	69 00 00 00 00 00 00 00 30 00 00 00 88 00 00 00	i.....0.....
00007ffc:c2cc2a20	00 00 00 00 09 00 00 00 33 00 00 00 00 00 00 00	.....3.....
00007ffc:c2cc2a30	62 00 00 00 44 00 00 00 02 00 00 00 69 00 00 00	b...D.....i...
00007ffc:c2cc2a40	00 00 00 00 30 00 00 00 88 00 00 00 00 00 00 00	.....0.....
00007ffc:c2cc2a50	ac 00 00 00 33 00 00 00 00 00 00 00 9d 00 00 00	.....3.....
00007ffc:c2cc2a60	44 00 00 00 02 00 00 00 69 00 00 00 00 00 00 00	D.....i.....
00007ffc:c2cc2a70	30 00 00 00 88 00 00 00 00 00 00 00 10 00 00 00	.....0.....
00007ffc:c2cc2a80	33 00 00 00 00 00 00 00 7e 00 00 00 44 00 00 00	3.....~...D...
00007ffc:c2cc2a90	02 00 00 00 69 00 00 00 00 00 00 00 30 00 00 00	.....i.....0...
00007ffc:c2cc2aa0	88 00 00 00 00 00 00 00 aa 00 00 00 33 00 00 00	.....D.....3...
00007ffc:c2cc2ab0	00 00 00 00 cd 00 00 00 44 00 00 00 02 00 00 00	.....D.....
00007ffc:c2cc2ac0	69 00 00 00 13 00 00 00 01 00 00 00 99 00 00 00	i.....

En pseudocódigo sería algo así:

```

01 jmp 3      (77 00 00 00 03 00 00 00)
02 ret        (99 00 00 00)
03 movrv 0b   (69 00 00 00 00 00 00 00 0b 00 00 00)
04 cmp_eq 0b  (33 00 00 00 00 00 00 00 0b 00 00 00)
05 jmp_ne 2   (44 00 00 00 02 00 00 00)

```

```

06 movrv 30    (69 00 00 00 00 00 00 00 30 00 00 00)
07 xor d2      (88 00 00 00 00 00 00 00 d2 00 00 00)
08 cmp_eq 95   (33 00 00 00 00 00 00 00 95 00 00 00)
09 jmp_ne 2    (44 00 00 00 02 00 00 00)

```

```

10 movrv 30    (69 00 00 00 00 00 00 00 30 00 00 00)
11 xor d6      (88 00 00 00 00 00 00 00 d6 00 00 00)
12 cmp_eq e6   (33 00 00 00 00 00 00 00 e6 00 00 00)
13 jmp_ne 2    (44 00 00 00 02 00 00 00)

```

....

Ya llegados a este punto, vemos que se repite la estructura, la idea es que comprobar si el tamaño de la cadena es 0b, si lo es, sigue, si no ret. Luego va cargando los dígitos de la cadena introducida, le realiza un xor y compara el valor, si correcto sigue, si no, pues termina.

Para extraer la flag, buscamos los xor y el valor comparado, operamos y obtenemos el valor original:

```

d2 ⊕ 95 => 47 G
d6 ⊕ e6 => 30 0
87 ⊕ d3 => 54 T
ea ⊕ b5 => 5F _
d4 ⊕ bc => 68 h
02 ⊕ 32 => 30 0
1b ⊕ 2b => 30 0
09 ⊕ 62 => 6b k
ac ⊕ 9d => 31 1
10 ⊕ 7e => 6e n
aa ⊕ cd => 67 g

```

Luego tenemos la flag => G0T\_h00k1ng

Probamos en el ejecutable parcheado:

```

./main_r1
Flag: G0T_h00k1ng
Flag correcta! :D
UAM{G0T_h00k1ng}

```

**UAM{7b02cd3d2d3cea80359cf600799413d3}**

@bicacaro